Perfect numbers

Assignment written by Julie Zelenski

Perfect numbers
An exhaustive algorithm
Observing the runtime
What is SimpleTest?
Practice with TIME_OPERATION
Digging deeper into testing
Streamlining and more testing
Mersenne primes and Euclid
Turbo-charging with Euclid
Warmup conclusions

This warmup task gives you practice with C++ expressions, control structures, and functions, as well as testing and debugging your code.

Throughout the writeup, we pose thought questions (in the highlighted yellow boxes) for you to answer. The starter project includes the file **`short_answer.txt`** (located under "Other Files" in the Qt Project pane). Edit this file to fill in your responses and submit it with your code. Your section leader will review your responses as part of grading. We'll be evaluating the sincerity and thoughtfulness of your reflection and reasoning; not a rigid "right/wrong" thing.

Perfect numbers

This exercise explores a type of numbers called *perfect numbers*. Before we jump into the coding, let's begin with a little math and history.

A *perfect number* is an integer that is equal to the sum of its proper divisors. A number's proper divisors are those positive numbers that evenly divide it, excluding itself. The first perfect number is **6** because its proper divisors are **1**, **2**, and **3**, and **1 + 2 + 3 = 6**. The next perfect number is **28**, which equals the sum of its proper divisors: **1 + 2 + 4 + 7 + 14**.

Perfect numbers are an interesting case study at the intersection of mathematics, number theory, and history. The rich history of perfect numbers is a testament to how much these numbers have fascinated humankind through the ages. Using our coding powers, we can explore different algorithmic approaches to finding these special numbers.

An exhaustive algorithm

One approach to finding perfect numbers is using an *exhaustive* search. This search operates by brute force, looping through the numbers one by one, and testing each to determine if it is perfect. Testing whether a particular number is perfect involves another loop to find those numbers which divide the value and add those divisors to a running sum. If that sum and the original number are equal, then you've found a perfect number!

Here is some Python code that performs an exhaustive search for perfect numbers:

```
def divisor_sum(n):
    total = 0
    for divisor in range(1, n):
        if n % divisor == 0:
            total += divisor
    return total

def is_perfect(n):
    return n != 0 and n == divisor_sum(n)

def find_perfects(stop):
    for num in range(1, stop):
        if is_perfect(num):
            print("Found perfect number: ", num)
        if num % 10000 == 0: print('.', end='',flush=True) # progress bar
    print("Done searching up to ", stop)
```

The Python code from above is re-expressed in C++ below. If your CS106A was taught in Python, comparing and contrasting these two may be a helpful way to start adapting to the language

Perfect numbers
An exhaustive algorithm
Observing the runtime
What is SimpleTest?
Practice with TIME_OPERATION
Digging deeper into testing
Streamlining and more testing
Mersenne primes and Euclid
Turbo-charging with Euclid
Warmup conclusions

differences. If instead your prior experience was in Java or Javascript, just sit back and enjoy how C++ already seems familiar to what you know!

```
/* The divisorSum function takes one argument `n` and calculates the
 * sum of proper divisors of `n` excluding itself. To find divisors
 * the loop iterates over all numbers from 1 to n-1, testing for a
 * zero remainder from the division using the modulus operator %
 *
 * Note: the C++ long type is a variant of int that allows for a
 * larger range of values. For all intents and purposes, you can
 * treat it like you would an int.
 */
long divisorSum(long n) {
    long total = 0;
    for (long divisor = 1; divisor < n; divisor++) {
        if (n % divisor == 0) {
            total += divisor;
        }
    }
    return total;
}

/* The isPerfect function takes one argument `n` and returns a boolean
 * (true/false) value indicating whether or not `n` is perfect.
 * A perfect number is a non-zero positive number whose sum
 * of its proper divisors is equal to itself.
 */
bool isPerfect(long n) {
    return (n != 0) && (n == divisorSum(n));
}

/* The findPerfects function takes one argument `stop` and performs
 * an exhaustive search for perfect numbers over the range 1 to `stop`.
 * Each perfect number found is printed to the console.
 */
void findPerfects(long stop) {
    for (long num = 1; num < stop; num++) {
        if (isPerfect(num)) {
            cout << "Found perfect number: " << num << endl;
        }
        if (num % 10000 == 0) cout << "." << flush; // progress bar
    }
    cout << endl << "Done searching up to " << stop << endl;
}
```

Observing the runtime

The starter project contains the C++ code above. It is given to you pre-written and works correctly. Look over the code and confirm your understanding of how it works. If you have questions or points of confusion, make a post on Ed to start a conversation or come by Lair or office hours.

We want you to first observe the performance of the given code. One simple approach for measuring runtime (how long it takes for the program to run to completion) is to simply run the program while keeping an eye on your watch or a clock. Open the starter project in Qt Creator and build and run the program. When you are prompted in the console to "Select the test groups you wish to run," enter 0 for "None" and the program will instead proceed with the ordinary **`main`** function. This **`main`** function does a search for perfect numbers across the range 1 to 40000. Watch the clock while the program runs and note the total elapsed time.

Perfect numbers
An exhaustive algorithm
Observing the runtime
What is SimpleTest?
Practice with TIME_OPERATION
Digging deeper into testing
Streamlining and more testing
Mersenne primes and Euclid
Turbo-charging with Euclid
Warmup conclusions

Q1. Roughly how long did it take your computer to do the search? How many perfect numbers were found and what were they?

You now have one timing result (how long it takes for a range of size 40000), but would need additional data to suss out the overall pattern. What would be ideal is to run the search several times with different search sizes (20000, 40000, 80000, etc.) and measure the runtime for each. To do this manually, you would edit **`main.cpp`**, change the argument to **`findPerfects`** and re-run the program while again watching the clock. Repeating for many different sizes would be tedious. It would be more convenient to run several time trials in sequence and have the program itself measure the elapsed time. The SimpleTest framework can help with this task, so let's take a detour there now.

What is SimpleTest?

In CS106B, you will use a unit-test framework called **`SimpleTest`** to test your code. This type of testing support will be familiar to you if you've been exposed to Java's **`JUnit`** or python **`doctest`**. (and no worries if you haven't yet, there will be much opportunity to practice with unit testing in CS106B!)

🛑 **Stop here and read our guide to testing** to introduce yourself to SimpleTest. For now, focus on use of **`STUDENT_TEST`**, **`EXPECT`**, **`EXPECT_EQUAL`**, and **`TIME_OPERATION`**, as these are the features you will use in Assignment 1.

Practice with `TIME_OPERATION`

Now that you know the basics of SimpleTest, you are ready to practice with it using the provided tests in **`perfect.cpp`**.

- Open the **`perfect.cpp`** file and scroll to the bottom. Review the provided test cases to see how they are constructed. One of the provided test shows a sample use of **`TIME_OPERATION`** to measure the time spent during a call to **`findPerfects`**.
- Run the program and when prompted to "Enter your selection," enter the number that corresponds to the option for the **`perfect.cpp`** tests. This will run the test cases from the file **`perfect.cpp`**. The SimpleTest window will open to show the results of each test. The given code should pass all the provided tests. This shows you what a successful sweep will look like. Note that the SimpleTest results window also reports the elapsed time for each **`TIME_OPERATION`**.

When constructing a time trial test, you will want to run several **`TIME_OPERATION`** across a set of different input sizes that allow the trend line to emerge. Selecting appropriate input sizes is a bit of an art form. If you choose a size that is too small, it can finish so quickly that the time is not measurable or gets lost in the noise of other activity. A size that is too large will have you impatiently waiting for it to complete. You are generally aiming for test cases that complete in 10-60 seconds. Note that a size that is just right for your friend's computer might be too small or too larger for yours, so you may need a bit of trial and error on your system to find a good range.

Set up an experiment by adding a **`STUDENT_TEST`** that does a single **`TIME_OPERATION`** of **`findPerfects`**. Try different sizes to determine the largest size which your computer can complete in around 60 seconds or so. Now edit your **`STUDENT_TEST`** to do four time trials on increasing sizes, doubling the size each time and ending with the final trial of that large size. The starter code includes some sample code showing how to use a loop to configure a series of time trials. This is a great technique to add to your repertoire! Run your time trials and write down the reported times.

Q2. Make a table of the timing results for **`findPerfects`** that you observed. (old-school table of text rows and columns is just fine)

Use the data from your table to work out the relationship between the search size and the

Perfect numbers
An exhaustive algorithm
Observing the runtime
What is SimpleTest?
Practice with TIME_OPERATION
Digging deeper into testing
Streamlining and more testing
Mersenne primes and Euclid
Turbo-charging with Euclid
Warmup conclusions

amount of time required. To visualize the trend, it may help to sketch a plot of the values (either by hand or using a tool like https://www.desmos.com/calculator).

You will find that doubling the input size doesn't take simply twice the time; the time goes up by a factor of 4. There appears a quadratic relationship between size and program execution time! Let's investigate why this might be.

> **Q3**. Does it take the same amount of work to compute `isPerfect` on the number 10 as it does on the number 1000? Why or why not? Does it take the same amount of work for `findPerfects` to search the range of numbers from 1-1000 as it does to search the numbers from 1000-2000? Why or why not?

That quadratic relationship means the algorithm is going to seriously bog down for larger ranges. Sure, the first four perfect numbers pop out in a jif (`6, 28, 496, 8128`) because they are encountered early in the search, but that fifth one is quite a ways off – in the neighborhood of 33 million. Working up to that number by exhaustive search is going to take l-o-o-ng time.

> **Q4**. Extrapolate from the data you gathered and make a prediction: how long will it take `findPerfects` to reach the fifth perfect number?

This is your first exposure to *algorithmic analysis*, a topic we will explore in much greater detail throughout the course.

As a fun aside, if you have access to a Python environment, you can attempt similarly-sized searches using the Python version of the program to see just how much slower an interpreted language (Python) is compared to a compiled language (C++). Check out the Python vs C++ Showdown we've posted to the Ed discussion forum.

Digging deeper into testing

Having completed these observations about performance, we move on to testing the code, which is one of the most important skills with which we hope you will come out of this class. Designing good tests and writing solid, well-tested code are skills that will serve you well for the rest of your computer science journey!

Here are some further explorations to do with SimpleTest:

- All perfect numbers are positive, there is no such thing as a negative perfect number; thus `isPerfect` should return false for any negative numbers. Will you need to make a special case for this or does the code already handle it? Let's investigate…
    - Look at the code and make a prediction about what happens if `isPerfect` is given a negative input.
    - Add a new `STUDENT_TEST` case that calls `isPerfect` on a few different negative inputs. The expected result of `isPerfect` for such inputs should be `false`.
        - A quick reminder that the tests that we give you in the starter code will always be labeled `PROVIDED_TEST` and you should never modify the provided tests. When you add tests of your own, be sure to label them `STUDENT_TEST`. This allows your grader to easily identify which tests are yours. We also prefer that your list your `STUDENT_TEST` cases first, ahead of the `PROVIDED_TEST` cases.
    - Run your new test cases to confirm that the code behaves as expected.
- Introduce a bug into `divisorSum` by erroneously initializing `total` to 1 instead of zero. Rebuild and run the tests again. This lets you see how test failures are reported.

> **Q5**. Do any of the tests still pass even with this broken function? Why or why not?

- Be sure to undo the bug and restore `divisorSum` to a correct state before moving on!

Streamlining and more testing

Perfect numbers
An exhaustive algorithm
Observing the runtime
What is SimpleTest?
Practice with TIME_OPERATION
Digging deeper into testing
Streamlining and more testing
Mersenne primes and Euclid
Turbo-charging with Euclid
Warmup conclusions

The exhaustive search does a lot of work to find the perfect numbers. However, there is a neat little optimization that can significantly streamline the process. The function **`divisorSum`** runs a loop from **`1`** to **`N`** to find divisors, but this is a tad wasteful, as we actually only need to examine divisors up to the square root of **`N`**. Each time we find a divisor, we can directly compute what the corresponding pairwise factor for that divisor (no need to search all the way up to N to find it). In other words, we can take advantage of the fact that each divisor that is less than the square root is paired up with a divisor that is greater than the square root. Take a moment to carefully think about why this is true and how you would rearrange the code to capitalize on this observation.

You are to implement a new function **`smarterSum`**:

```
long smarterSum(long n)
```

that produces the same result as the original **`divisorSum`** but uses this optimization to tighten up the loop. You may find it helpful to use the provided **`divisorSum`** implementation as a starting point (that is, you may choose to copy-paste the existing code and make tweaks to it). Be careful: there are some subtle edge cases that you may have to handle in the adapted version that were not an issue in the original.

The C++ library function **`sqrt`** can be used to compute a square root.

After adding new code to your program, **your next step is to thoroughly test that code** and confirm it is bug-free. Only then should you move on to the next task. Having just written **`smarterSum`**, now is the time for you to test it.

Because you already have the vetted function **`divisorSum`** at your disposal, a clever testing strategy uses **`EXPECT_EQUAL`** to confirm that the result from **`divisorSum(n)`** is equal to the result from **`smarterSum(n)`**. Rather than pick any old values for **`n`**, brainstorm about which values would be particularly good candidates. As an example, choosing **`n=25`** (which has an square root that is an integer value) could confirm there is no off-by-one issue on the stopping condition of your new loop. **Add at least 3 new student test cases for `smarterSum`.**

> **Q6**. Explain your testing strategy for **`smarterSum`** and how you chose your specific test cases that lead you to be confident the function is working correctly.

Now with confidence in **`smarterSum`**, implement the following two functions that build on it:

```
bool isPerfectSmarter(long n)

void findPerfectsSmarter(long stop)
```

The code for these two functions is *very similar* to the provided **`isPerfect`** and **`findPerfects`** functions, basically just substituting **`smarterSum`** for **`divisorSum`**. Again, you may copy-paste the existing implementations, and make small tweaks as necessary.

Now let's run time trials to see just how much improvement we've gained.

- Add a **`STUDENT_TEST`** that uses a sequence of **`TIME_OPERATION`** to measure **`findPerfectsSmarter`**. Set up 4 runs on increasing sizes, doubling the size each time, and ending with a run on the maximum size your program can complete in around 60 seconds using this improved algorithm. Given the smarter algorithm, these sizes will be larger than you used for the original version.

> **Q7**. Record your timing results for **`findPerfectsSmarter`** into a table.

Our previous algorithm grew at the rate **`N^2`**, while this new version is $N\sqrt{N}$, since we only have to inspect $\sqrt{N}$ divisors for every number along the way. If you plot runtimes on the same graph as before, you will see that they grow much less steeply than the runtimes of the original algorithm.

> **Q8**. Make a prediction: how long will **`findPerfectsSmarter`** take to reach the fifth perfect number?

Perfect numbers
An exhaustive algorithm
Observing the runtime
What is SimpleTest?
Practice with TIME_OPERATION
Digging deeper into testing
Streamlining and more testing
Mersenne primes and Euclid
Turbo-charging with Euclid
Warmup conclusions

Mersenne primes and Euclid

Back to story time: in 2018, there was a rare finding of a new Mersenne prime. A *Mersenne number* is a number that is one less than a power of two, i.e., of the form $2^k - 1$ for some integer k. A *prime number* is one whose only divisors are 1 and the number itself. A Mersenne number that is prime is called a *Mersenne prime*. The Mersenne number $2^5 - 1$ is 31 and 31 is prime, making it a Mersenne prime.

Mersenne primes are quite elusive; the most recent one found is only the 51st known and has almost 25 million digits! Verifying that the found number was indeed prime required almost two weeks of non-stop computing. The quest to find further Mersenne primes is driven by the Great Internet Mersenne Prime Search (GIMPS), a cooperative, distributed effort that taps into the spare computing cycles of a vast community of volunteer machines. Mersenne primes are of great interest because they are some of the largest known prime numbers and because they show up in interesting ways in games like the Tower of Hanoi and the wheat and chessboard problem.

Back in 400 BCE, Euclid discovered an intriguing one-to-one relationship between perfect numbers and the Mersenne primes. Specifically, if the Mersenne number $2^k - 1$ is prime, then $2^{(k-1)} * (2^k - 1)$ is a perfect number. The number 31 is a Mersenne prime where k = 5, so Euclid's relation applies and leads us to the number $2^4 * (2^5 - 1) = 496$ which is a perfect number. Neat!

Those of you enrolled in CS103 will appreciate this lovely proof of the Euclid-Euler theorem.

Turbo-charging with Euclid

The final task of the warmup is to leverage the cleverness of Euclid to implement a blazingly-fast alternative algorithm to find perfect numbers that will beat the pants off of exhaustive search. Buckle up!

Your job is to implement the function:

```
long findNthPerfectEuclid(long n)
```

The exhaustive search algorithm painstakingly examines every number from 1 upwards, testing each to identify those that are perfect. Taking Euclid's approach, we instead hop through the numbers by powers of two, checking each Mersenne number to see if it is prime, and if so, calculate its corresponding perfect number. The general strategy is outlined below:

1. Start by setting `k = 1`.
2. Calculate `m = 2^k - 1` (Note: C++ has no exponentiation operator, instead use library function `pow`)
3. Determine whether `m` is prime or composite. (Hint: a prime number has a `divisorSum` and `smarterSum` equal to one. Code reuse is your friend!)
4. If `m` is prime, then the value $2^{(k-1)} * (2^k - 1)$ is a perfect number. If this is the `nth` perfect number you have found, stop here.
5. Increment `k` and repeat from step 2.

The call `findNthPerfectEuclid(n)` should return the `n`th perfect number. What will be your testing strategy to verify that this function returns the correct result? You may find this table of perfect numbers to be helpful. One possibility is a test case that confirms each number is perfect according to your earlier function, e.g. `EXPECT(isPerfect(findNthPerfectEuclid(n)))`.

Note: The `findNthPerfectEuclid` function can assume that all inputs (values of `n`) will be positive (that is, greater than 0). In particular, this means that you don't have to worry about negative values of `n` or the case where `n` is zero.

Add at least 4 new student test cases of your own to verify that your `findNthPerfectEuclid` works correctly.

Perfect numbers
An exhaustive algorithm
Observing the runtime
What is SimpleTest?
Practice with TIME_OPERATION
Digging deeper into testing
Streamlining and more testing
Mersenne primes and Euclid
Turbo-charging with Euclid
Warmup conclusions

Q9. Explain how you chose your specific test cases and why they lead you to be confident **`findNthPerfectEuclid`** is working correctly.

A call to **`findNthPerfectEuclid(5)`** should near instantaneously return the fifth perfect number. Woah! Quite an improvement over the exhaustive algorithm, eh? But if you try for the sixth, seventh, eighth, and beyond, you can run into a different problem of scale. The super-fast algorithm works in blink of an eye, but the numbers begin to have so many digits that they quickly exceed the capability of the **`long`** data type (**`long`** can hold a maximum of 10 to 20 digits depending on your system. When a number gets too large, the value will unexpectedly go negative — how wacky is that? Take CS107 to learn more)

So much for the invincibility of modern computers… As a point of comparison, back in 400 BC, Euclid worked out the first eight perfect numbers himself — not too shabby for a guy with no electronics!

Warmup conclusions

Hooray for algorithms! One of the themes for CS106B is the tradeoffs between algorithm choices and program efficiency. The differences between the exhaustive search and Euclid's approach is striking. Although there are tweaks (such as the square root trick) that will improve each algorithm relative to itself, the biggest bang for the buck comes from starting with a better overall approach. This result foreshadows many of the interesting things to come this quarter.

Before moving on to the second part of the assignment, confirm that you have completed all tasks from the warmup. You should have answers to questions Q1 to Q9 in **`short_answer.txt`**. You should have implemented the following functions in **`perfect.cpp`**

- **`smarterSum`**
- **`isPerfectSmarter`**
- **`findPerfectsSmarter`**
- **`findNthPerfectEuclid`**

as well as added the requisite number of tests for each of these functions. Your code should also have thoughtful comments, including an overview header comment at the top of the file.

In the header comment for **`perfect.cpp`**, share with your section leader a little something about yourself and to offer an interesting tidbit you learned in doing this warmup (be it something about C++, algorithms, number theory, how spunky your computer is, or some other neat insight).
